Solid State Drive

By
Shaun Steele

Senior Project
ELECTRICAL ENGINEERING DEPARTMENT
California Polytechnic State University
San Luis Obispo
2017

Contents

List of Tables and Figures

Figures:

Tables:

Abstract

This project documents the design and implementation of a solid state drive (SSD). SSDs are a non-volatile memory storage device that competes with hard disk drives. SSDs rely on flash memory, a type of non-volatile memory that is electrically erased and programmed. The appeal of SSDs lies in the fact that they allow a fast, reliable, and durable memory storage device. The goal of this project is to have a working external SSD built from scratch.

Chapter I. Introduction

The computer hardware industry is a rapidly growing field that has new advancements each day. As the entire technology industry continues improving, more and more data is produced. While the data remains unused, it must be stored, which is where mass storage devices come into play. The two types of mass storage devices used today are hard disk drives (HDDs) and solid state drives (SSDs).

HDDs currently hold the position of the most popular mass storage device, as their price per gigabyte (GB) is very low at about four cents [1]. Through discussion with a Western Digital[1] employee in Fall 2016, the only reason they are not selling more HDDs is because they cannot manufacture them fast enough.

The SSD follows the HDD in popularity, mainly due to its cost of about twenty to fifty cents per GB [1], a price five to ten times more than HDDs. SSDs are faster, more physically durable, and can be condensed into smaller form factors. SSDs typically have memory access speeds of about ten to one-hundred microseconds, while HDDs' memory access speeds are about five to ten milliseconds. The durability of SSDs is higher than HDDs as there are no moving parts to account for. HDDs store data on a magnetic rotating disk, which is accessed through use of a magnetic head attached to an actuator. Typical options for the size of HDDs are 3.5 inches or 2.5 inches, while SSDs can be found in sizes of 2.5, 1.8 and 1.0 inches. [2]

This project explores the design and implementation of building an external SSD from scratch. I was motivated by a desire to learn about how the technology that powers SSDs. The ability to understand an SSD as a system not only requires knowledge learned from Cal Poly's electrical engineering curriculum, but also requires knowledge of many topics not covered in class.

[1] Western Digital is one of the largest HDD manufacturers in the world.

Chapter II. Background

Hardware

Flash memory is the fundamental building block of SSD. The specific type of flash memory used in SSDs is NAND flash. NAND flash is a very dense layout of byte-addressable non-volatile memory that is illustrated in Figure 1. It is given the name NAND as its transistors are in series as opposed to parallel, much like a CMOS NAND gate. Floating Gate Metal Oxide Semiconductor Field Effect Transistors (FGMOS) in parallel are the basis for NOR flash memory.

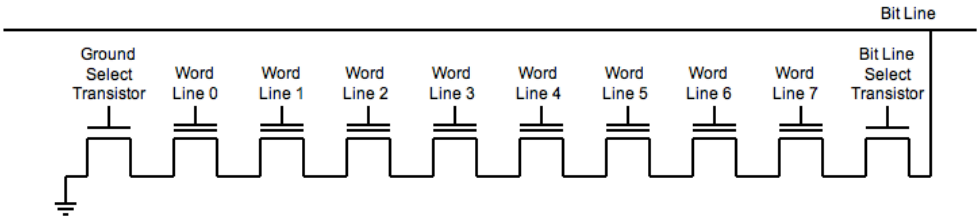

Figure 1. Typical NAND Flash Schematic [3]

NAND flash technology allows for a high memory density, and nonvolatile data storage. The key component of flash memory is the FGMOS.

The FGMOS has two gates, as opposed to a MOSFET's one gate. The control gate is connected to an external circuit. The floating gate earns its name by having no connections to external conductors. Figure 2 depicts an accurate cross section of an FGMOS.

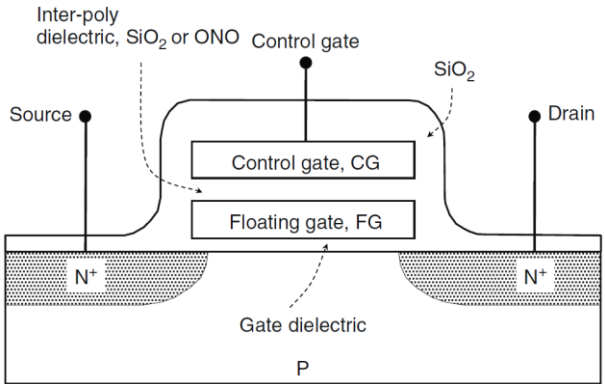

Figure 2. Typical Floating-gate MOSFET Cell

The floating-gate acts as a memory cell, which can trap charge by electrons tunneling through the oxide layer (gate dielectric) or injected when a voltage is applied to the control gate that is greater than the threshold voltage. [4]

Reading and writing for flash memory look a little different from other memory types like Dynamic and Static Random Access Memory (DRAM and SRAM). A read is still a read, but a traditional write is called either an 'Erase' or 'Program'. This naming convention eliminates the idea of overwriting data, or rewriting new data to memory. The origin of the convention stems from the fact that the FGMOS can be in one of two states. Without any data being 'written' to the transistor, the FGMOS will be in an erased state, or when read from, a logic high. Writing data to the transistor results in a programmed state.

When a FGMOS is in its erased state, it has a low threshold voltage V_{T0}. When there is charge in the

floating-gate, the threshold voltage increases to V_T. This change in threshold voltage ΔV_T is depicted Equation 1 with relation to the floating-gate charge Q_{FG} and floating-gate to control-gate capacitance C_{FG-CG}.

$$\Delta V_T = V_T - V_{T0} = \frac{Q_{FG}}{C_{FG-CG}}$$

The effect of these V_Ts on FGMOS current is illustrated in Figure 3. Reading occurs by applying a voltage in between the two threshold voltages V_{RD}, and detecting the current. If there is current, the transistor is a logic high or 'erased'. If there is no current, the transistor is a logic low or 'programmed'.

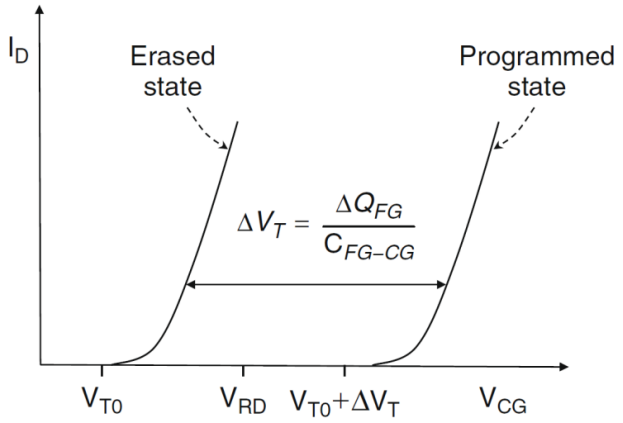

Figure 3. FGMOS Drain Current vs. Control-Gate Voltage Characteristic for Erased and Programmed States

As mentioned previously, there is no overwriting memory to the cell. To write new data to a cell, a cell must be programmed or erased. This action has two effects. The first is the expected change in logic states, and the second is wear. Wear occurs within the gate oxide layer. As electrons are tunneled or injected through the oxide layer, the layer deteriorates. The wear decreases the difference in threshold voltage window with time due to electron drift. The concept of wear gives each FGMOS a limited Program/Erase endurance, the amount of times a FGMOS may cycle through being programmed and erased while maintaining a distinct difference in threshold voltages.

To make a useful SSD, the drive must not only be able to read, program, and erase from the flash memory, but also extend the lifetime of its memory by spreading the wear evenly among all memory cells. The idea of spreading the wear across all FGMOS cells is called wear-leveling. All modern SSDs have wear-leveling algorithms to extended endurance of the drive.

Firmware

A typical personal computer (PC) cannot communicate with flash memory without the use of a microcontroller unit (MCU) or processor loaded with firmware containing a communication interface to the PC and driver for interacting with the flash memory. A very simple block diagram depicting the inner layout of an SSD is shown in Figure 4.

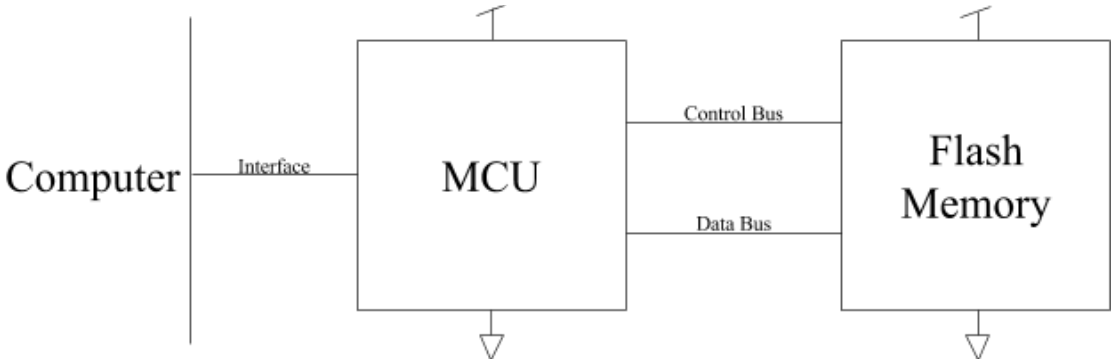

Figure 4. Level 0 Solid State Drive Block Diagram

Historically, PCs have been designed to interface with mass storage for HDDs. This poses a problem for flash memory, as flash memory is completely different from magnetic disks in terms of how data is accessed, and organized. Many common file systems like File Allocation Table (FAT) and New Technology File System (NTFS) have been created for use on HDDs, and are extremely inefficient for flash. The solution to using flash memory efficiently with common file systems is called the Flash Translation Layer (FTL).

How the PC's operating system interacts with memory must be explored before understanding the necessity of the FTL. Addressing memory on disk can happen two ways. The first way involves using specific fields for the cylinder, head, and sector to determine where data is located. Figure 5 illustrates the organization of cylinder-head-sector in an HDD. The newer method uses an idea that converts cylinder-head-sector into a Logical Block Address (LBA). An LBA is an address that abstracts the physical component of addressing memory, and is determined based on a calculation using the cylinder-head-sector within HDDs. The LBA makes interfacing with flash a little bit easier as it is just an address and not a physical description.

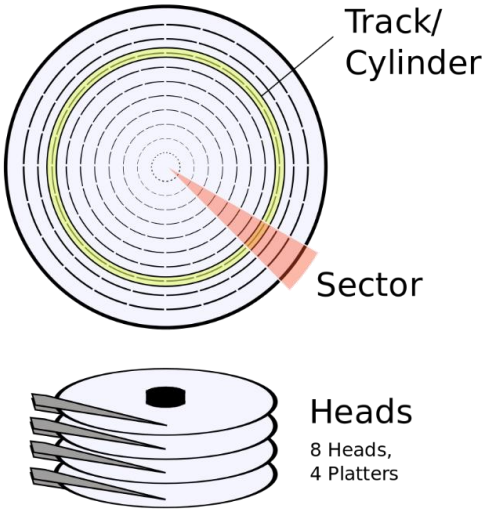

Figure 5. Cylinder-Head-Sector Layout of an HDD [5]

For SSDs, the FTL translates an LBA to a Physical Block Address (PBA). Upon writing to an SSD at an unused LBA, the flash memory will map the logical address to a physical address for the data to be stored. To read data, the operating system will look to the LBA, the translation layer will find the PBA,

and then allow the operating system to read the data. Writing new data to an already used LBA requires an algorithm that not only preserves data, but also levels wear.

The algorithm that is used is listed:

1. Mark the current PBA as stale
2. Generate a new PBA
3. Link current LBA to new PBA
4. Write data to current LBA, which is the new PBA
5. When ready, collect garbage

This algorithm is developed for this project. What this algorithm does is associate a new PBA with the desired LBA. This leaves un-writeable data left within the flash memory. This extra data is taken care of during garbage collection. Garbage collection is a technique used to erase all memory cells at once, to allow use from the maximum amount of memory at a time.

The last step to create an SSD is choosing the protocol for communication between the computer and the controller. Common methods that are used are derivations of Serial AT Attachment (SATA) and Universal Serial Bus (USB).

Chapter III. Requirements

The deliverable for this project is an SSD. This entails the drive must be recognizable to an operating system as a mass storage device, physically connect to a computer, and store files. Table I shows the specification goals for this project.

Table 1. SSD Specifications

Specification Number	Specification	Justification
1.	System uses USB protocol for communication and power.	The SSD shall remain external to the computer. USB is chosen, as it is easy to use and available on most operating systems today.
2.	System has a storage capacity of 16GB or greater.	As storage capacity increases, usefulness and price increase. 16GB is a good balance between usefulness and price and availability of parts.
3.	System consists of at least an MCU and memory integrated circuit (IC).	As shown in Figure 4, the basic components of an SSD are the controller and memory IC.
4.	Memory IC is non-volatile NAND flash memory.	NAND flash memory is what makes an SSD a storage device that is comparable with HDDs. A memory device without flash memory is not an SSD.

Specification 1 in Table I covers the communication protocol between an external SSD and the host computer. Additional reasons for this specification include its documentation and power management. Thorough documentation on how to use USB is available to the public, making development and implementation of the protocol possible. USB is also low power. While operating in bus-powered mode, USB will allow the host to provide a maximum of 100 mA.

Flash memory is expensive, and chips with higher storage densities are difficult to acquire in prototype quantities. For this project, one, two, or four flash memory ICs can be used to reach a 16GB capacity easily, defining Specification 2. Any increase in storage capacity increases the difficulty of purchasing memory ICs that are cost efficient in small quantities.

The basic components of an SSD are the MCU and flash memory IC. This project must utilize these two components in order to be considered an SSD.

SSDs require the use of flash memory as it is non-volatile. An SSD without flash memory either is an HDD, or cannot store data. This project shall utilize NAND flash memory as it is denser than NOR flash memory, thus providing more capacity.

Now that the SSD specifications are defined, a system can be designed and implemented.

Chapter IV. Plan

Figures 6, 7, and 8 are Gantt charts that illustrate the project's plan for completion. Fall quarter consists mainly of planning the project and researching how the idea for the project compares with other similar devices on the market. During EE460, no design or implementation took place, as per instruction of the professor.

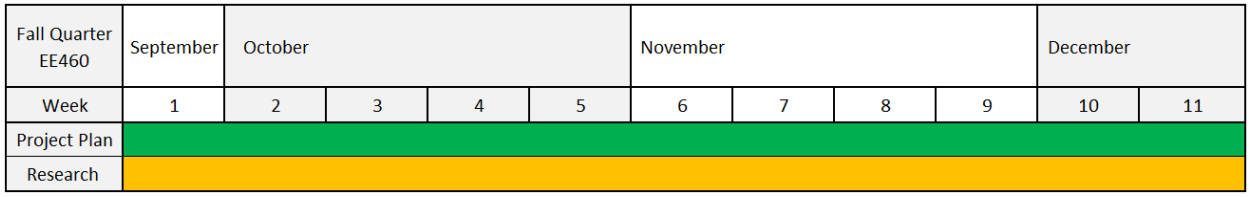

Figure 6. Fall Quarter Gantt Chart Plan

Figures 7 and 8 display the Gantt charts of the project plan and the actual outcome for Winter Quarter. The differences lie in the proof of concept design. To design a general SSD system, a great understanding is necessary. The projected time to learn the ins and outs of a basic SSD was too short, resulting in a delay for the completion of the proof of concept design.

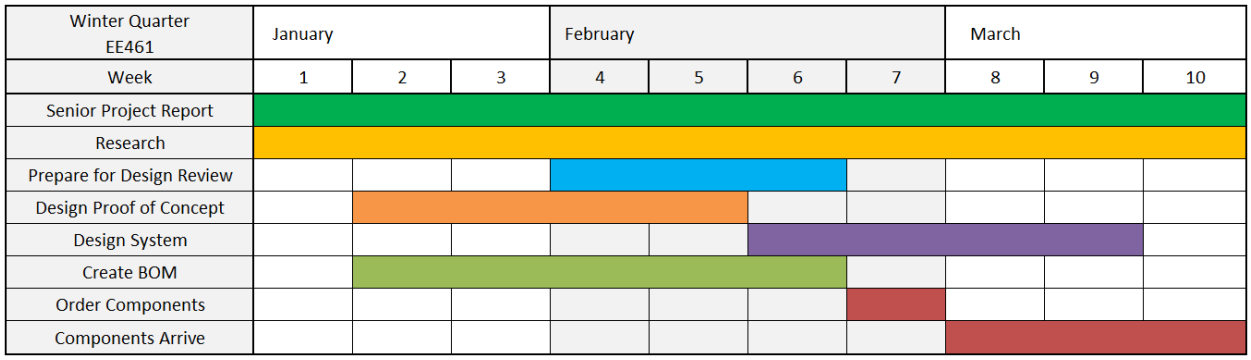

Figure 7. Winter Quarter Planned Gantt Chart

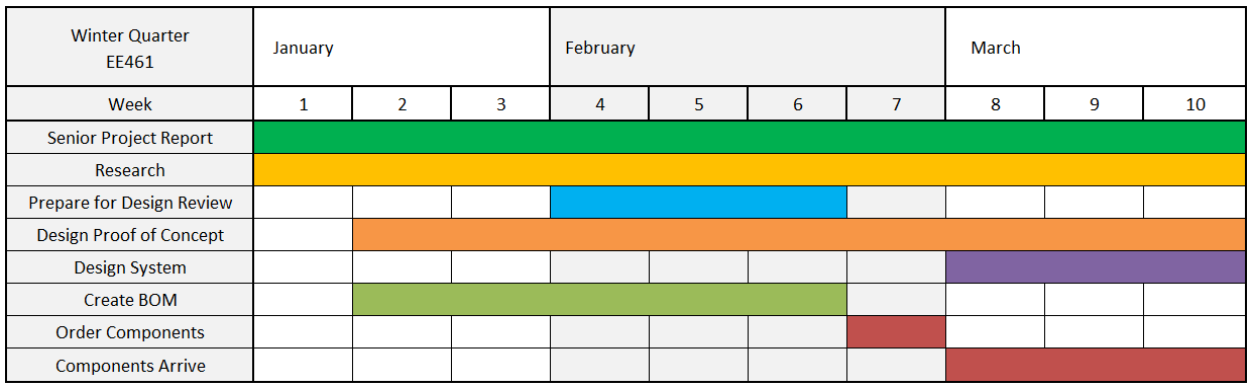

Figure 8. Winter Quarter Gantt Chart Outcome

Figures 9 and 10 display the Gantt charts of the project plan and outcome for Spring Quarter. Due to the delay and difficulties in designing the proof of concept, the final system design is incomplete, leaving the proof of concept the deliverable for this project.

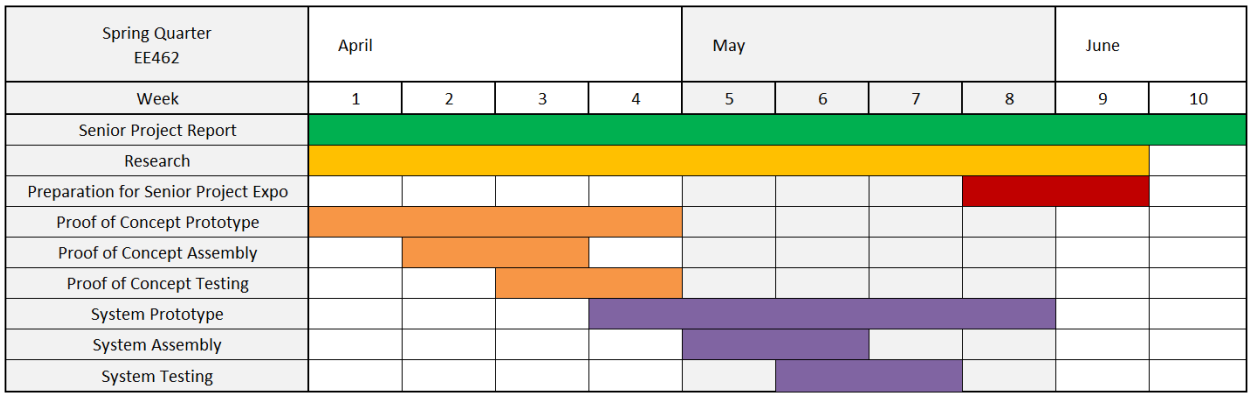

Figure 9. Spring Quarter Planned Gantt Chart

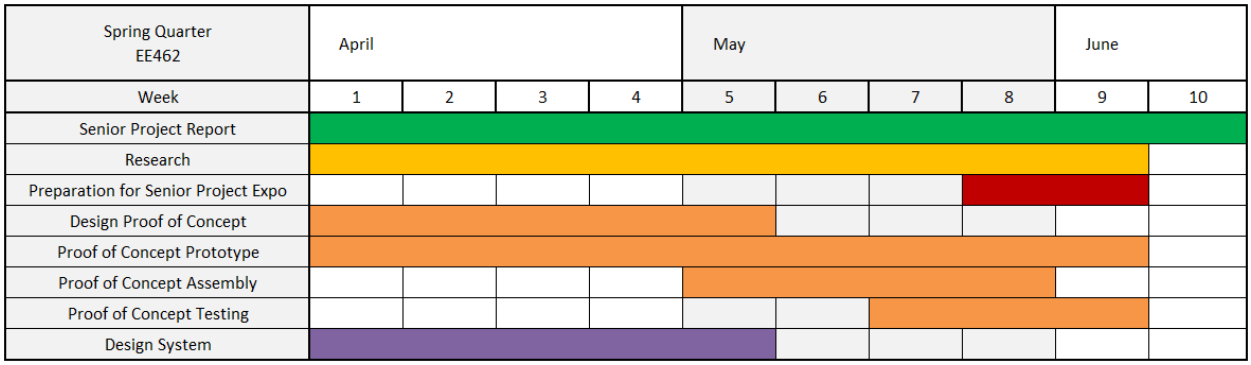

Figure 10. Spring Quarter Gantt Chart Outcome

Chapter V. Design

This project's design is broken into two parts: hardware and firmware. The hardware design consists of component selection of the fundamental blocks that make an SSD. Tables 2 and 3 list all chosen components. It is broken up into two sections: primary components and secondary components. The primary components are pieces that make up the proof of concept system, and the secondary components are for the final system. Due to the complex nature of the system, this project only uses the primary components.

Table 2. Primary Component Selection

No.	Name	Description	Distributor	Qty.	Price	Extended Price
1	Silicon Labs C8051F340-GQ	8051 C8051F34x Microcontroller IC, 8-Bit, 48MHz, 64KB Flash, 48-QFP	Digikey	1	$8.12	$8.12
2	Toshiba Semiconductor and Storage TC58NVG3S0FTA00	Flash – NAND Memory IC, 8Gb, Parallel, 25ns, 48-TSOP	Digikey	1	$10.46	$10.46

Table 2 lists the primary components used in this project. Component 1 is the MCU chosen for the system, and Component 2 is the flash memory chosen for the proof of concept system. The relatively small memory capacity of Component 2 makes it a good testing memory IC. The primary purpose of this component is to become familiar with flash memory EEPROMs, and write a simple driver to be scaled for a larger memory device.

Component 1 is the MCU chosen for this system. This component is an 8051 8-bit ARM microcontroller that uses the Keil c51 Toolchain [7] This MCU is chosen as it is has two desired features. The C8051F340 has USB2.0 Full-Speed supporting circuitry built into the IC, controlling the subsystem with multiple special function registers. The other desired feature is the number of general-purpose pins (GPIO) the device offers. It has 40 GPIO pins that are all configurable with an on-chip crossbar connecting all non-USB pins to port latches. This number of pins allows for up to four 48-pin thin small-outline packaged (TSOP) flash memory ICs to be used, along with an 8-pin Serial Peripheral Interface (SPI) RAM.

Table 3. Secondary Component Selection

No.	Name	Description	Distributor	Qty.	Price	Extended Price
3	Cypress Semiconductor Corp FM25W256-G	FRAM (Ferroelectric RAM) Memory IC, 32KB, SPI, 8-SOIC	Digikey	1	$5.81	$5.81
4	Micron Technology Inc. MT29F64G08CBABAWP:B TR	Flash-NAND Memory IC, 64Gb, Parallel, 48-TSOP	Digikey	4	$7.75	$31.00

For future work, Components 3 and 4 could be used to reach the goal of exceeding a 16GB storage capacity. Component 4 is a higher density flash memory chip with 64Gb or 8GB of storage. Four of these will create a drive of 32GB. To make use of a larger capacity than 1GB, the system needs much more memory, causing a demand for more RAM than what is available on the MCU.

More RAM is achieved by using Component 3. The firmware design section discusses how adding 32KB

of RAM improves the system's performance.

The overall firmware design process consisted of completing two main goals. Communicating with an operating system that the SSD is a storage system is the first goal, and storing data on the flash memory is the second goal. Specification 1 states the system must communicate using USB for communication and power, and the MCU chosen has USB2.0 support circuitry. This mix makes it natural to choose the USB Mass Storage Class for communication.

The USB Mass Storage Device class (MSD) is a USB interface that uses Bulk-Only Transport endpoints. Most storage devices, like USB flash drives, typically use USB MSD as their communication protocol. Small Computer Systems Interface (SCSI) is typically used in conjunction with USB MSD. Multiple protocols lead to multiple layers necessary for a storage system.

A mass storage device application note is provided for the Silicon Labs C8051F34X devices in [6]. The application note is a reference design for using USB MSD with the C8051F34X MCU with a Compact Flash or SD/MMC memory card. Figure 11 displays the USB Stack that the application note references. This project takes the stack and modifies it to fit the application of a flash memory IC.

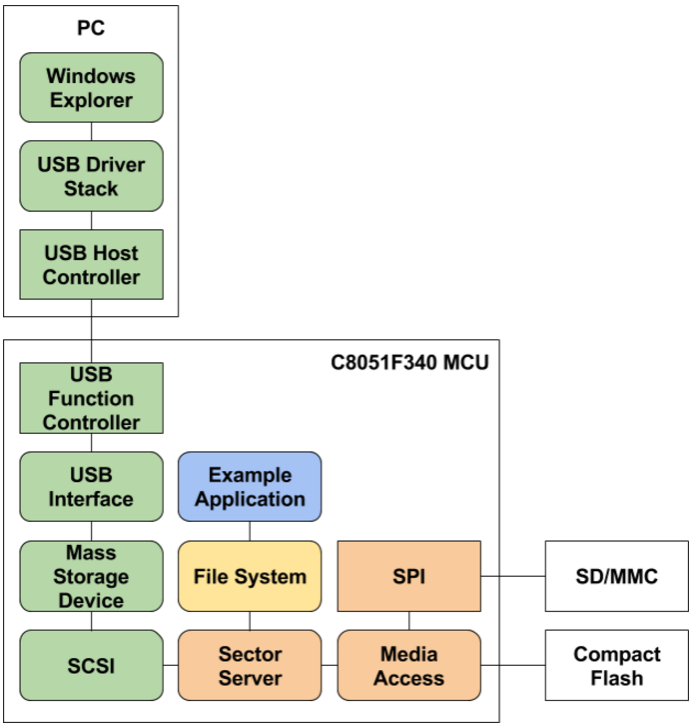

Figure 11. USB MSD Reference Design Application Note

This project removes the Example Application, SPI, and Media Access blocks from the application note, as they do not serve any purpose in this project. The new stack for the proof of concept includes the Flash Memory IC driver and FTL, both illustrated in Figure 12. The overall system stack is similar to the proof of concept, only adding three more Flash Memory ICs with connections to the Flash Memory IC Driver.

The file system used in the application note stack is a FAT16 file system. This file system has a capacity of 4GB. To increase the capacity of the file system, a new file system must be chosen. The chosen file system FAT32, which has 32GB capacity—a size that fits the specification of at least 16GB capacity.

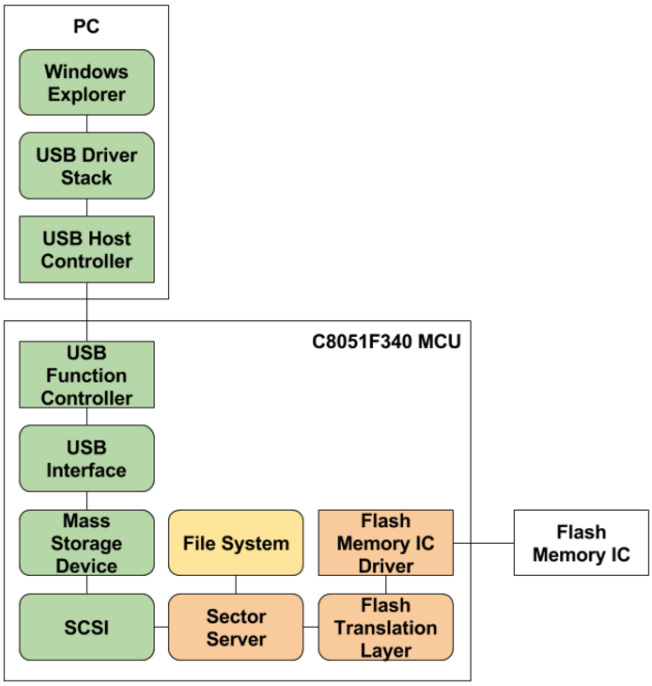

Figure 12. SSD USB MSD Stack

Chapter VI. Test Plans

The system is tested by validating that the SSD powers on, is recognized by the operating system as a USB Mass Storage Device, and can read and write files to the drive. Before this system test is implemented, each component and their interconnections with other components need to be checked. This chapter explores the test procedures used in this project.

Microcontroller Programming

Before testing the flash memory, the microcontroller-programming environment needs to be set up and verified. Because a Silicon Labs IC is used, the Silicon Labs IDE Simplicity Studios V4 is used, as it provides a programming and debugging environment. To actually program the C8051F340, the ToolStick340PP programming adapter and daughter cards are used. The C8051F340 is placed in a socket on the daughter card, and can be tested by soldering header pins to the daughter card. These pins also allow for connection to other peripherals using wires and a breadboard.

Flash Testing Environment

The flash memory IC chosen is a 48-pin surface mount component. To test the component on a breadboard, a 48-pin TSOP to 48-pin DIP socket is used. With wires and a breadboard, the flash memory IC connects to the C8051F340, allowing for testing to commence.

The next tests cover testing the driver written for controlling the flash memory IC. The first test checks to see if there is a sign of life from the memory IC upon power up. The power on/off sequence application note on page 61 of [8] displays how the flash memory powers on. The Ready/Busy pin determines if the flash memory IC successfully powers on. Figure 13 is the power on/off sequence of the flash memory IC. The Ready/Busy pin transitioning (noted by the red circle) to a logical high tells the MCU that the memory IC is ready to start operations.

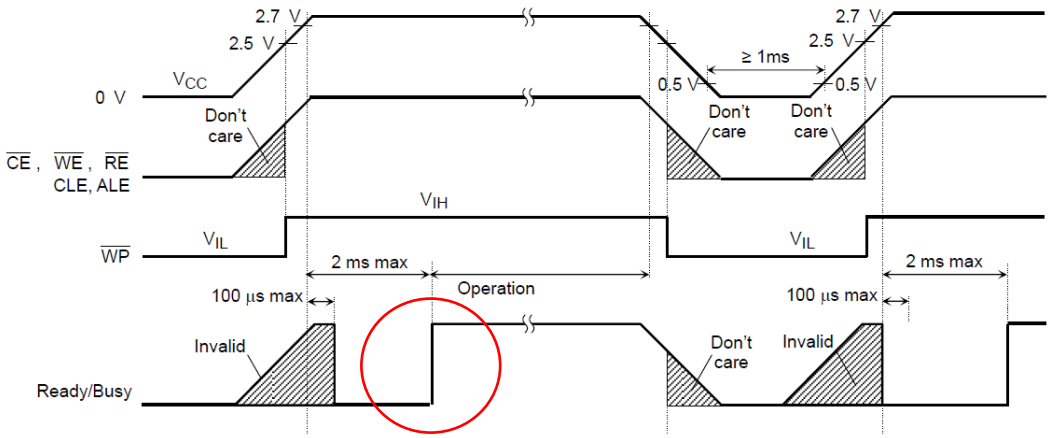

Figure 13. TC58NVG3S0FTA00 Power On/Off Sequence

The next test for the flash memory IC tests the Command, Address Input, and Input and Output Latching macros of the driver. An ID Read command is sent to the memory IC with an address of 0x00. Figure 14 illustrates the ID Read timing diagram of the memory IC, with the command operation in green, address input operation in blue, and output latches in orange. A successful test results in the MCU reading the values 0x98, 0xD3, 0x90, 0x26, and 0x76. Once the driver's macros are tested, the functions of Page Read, Page Program, and Block Erase may occur.

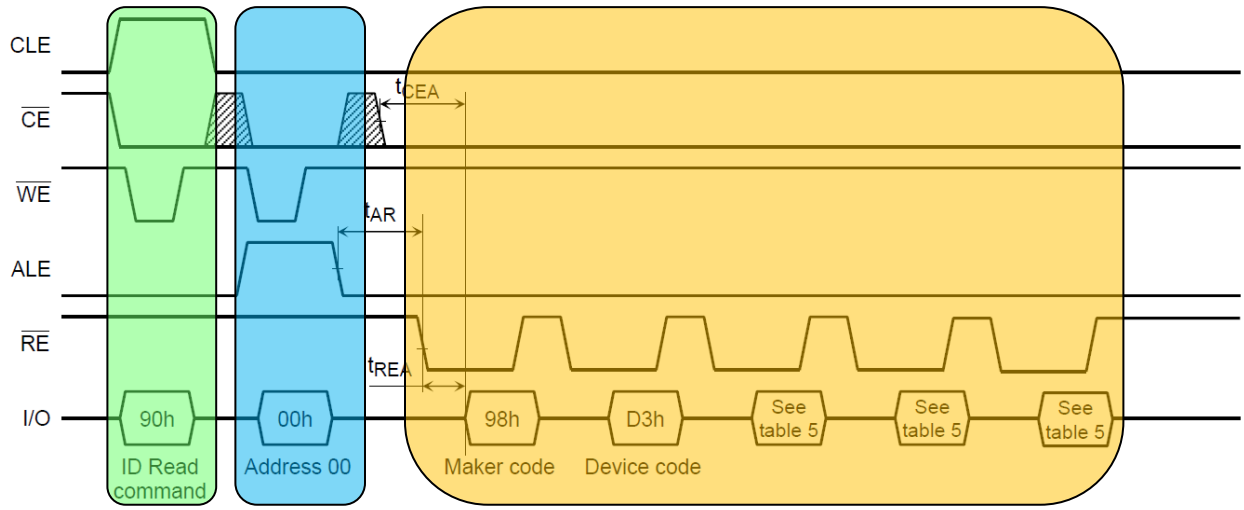

Figure 14. TC58NVG3S0FTA00 ID Read Timing Diagram

The test sequence for the Page Read, Page Program, and Block Erase subroutines consists of the following steps:

1. Program random data to a page at a random address and verify its status
2. Send a read command and use address previously programmed and check data validity
3. Erase the block at the address previously used and verify status
4. Read data from the same address and confirm the data is erased

This sequence determines if the system can perform the read and write operations that the operating system requires being a properly functioning SSD.

The next step for testing occurs on a software level where the FTL must properly read, program, and erase based on what functions the Sector Server layer calls. A virtual programming environment with fake data inputs and function calls is used to test the FTL. Each subroutine in the FTL is tested for validity that it performs as expected. Once FTL testing is completed, the proof of concept can be tested.

Testing the proof of concept starts by plugging the USB connector into a computer. Tests include the operating system noticing the device, recognizing the device as a USB Mass Storage Device, and formatting the drive, which then leads to reading and writing files to the drive. If the proof of concept passes these tests, the system is expanded to reach the specifications, and retested following the test procedure described in this chapter.

Chapter VII. Development and Construction

The development for this project follows the flow of the test plan. The construction of the proof of concept includes:

1. Solder headers to Toolstick340PP daughter card
2. Insert C8051F340 into daughter card socket
3. Insert TC58NVG3S0FTA00 into 48-pin socket
4. Connect daughter card and 48-pin socket using the a breadboard and wires following the schematic in Appendix B

The proof of concept is shown in Figure 15, where the Toolstick340PP and C8051F340 are in green, TC58NVG3S0FTA00 and 48-pin socket is in yellow, and the physical USB connector is in blue.

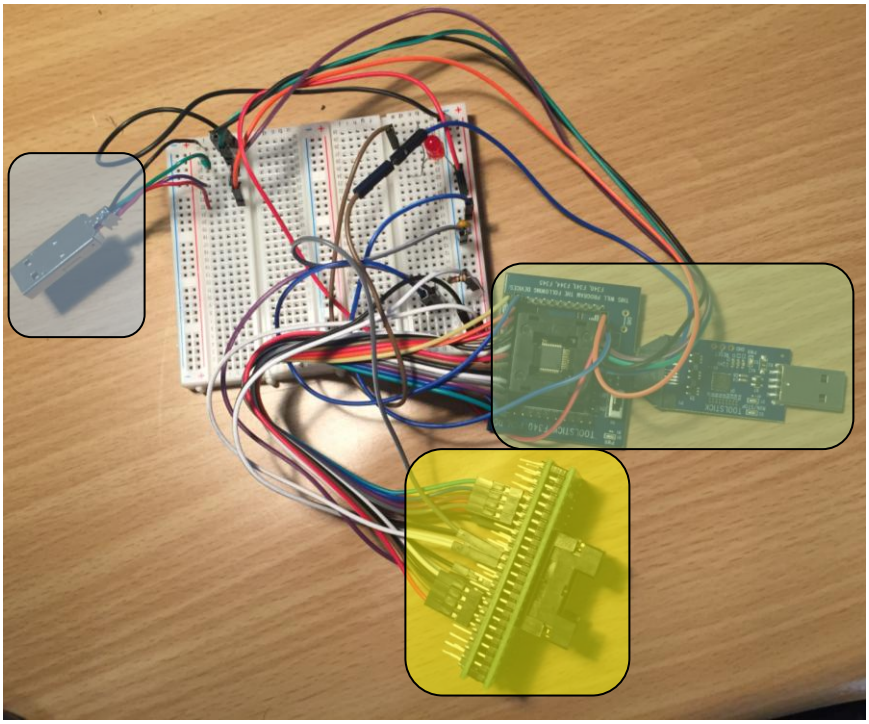

Figure 15. Proof of Concept Construction

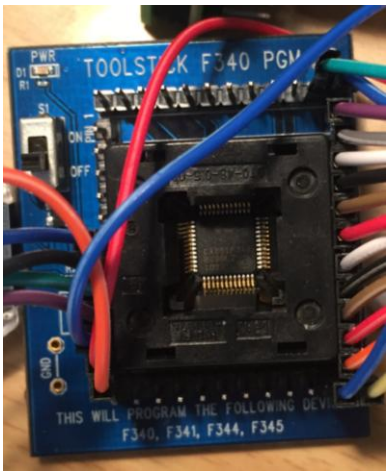

Figure 16. C8051F340 in Daughter Card Socket

Figures 16 and 17 show close-ups of the C8051F340 and TC58NVG3S0FTA00 in their sockets to give an understanding how the ICs are utilized.

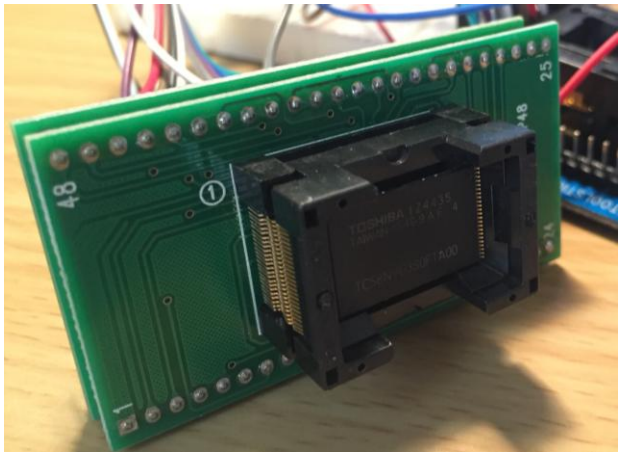

Figure 17. TC58NVG3S0FTA00 in its 48-Pin Socket

The development of the driver is composed of writing the essential functions for reading, programming, and erasing that are called by the FTL. Each of these functions follow the timing diagrams shown in the TC58NVG3S0FTA00 datasheet [8], also shown in Figures 18, 19, and 20.

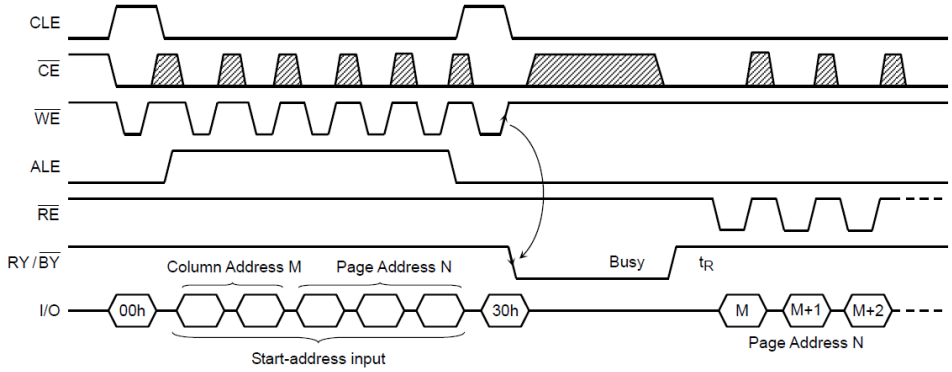

Figure 18. Read Mode Timing Diagram

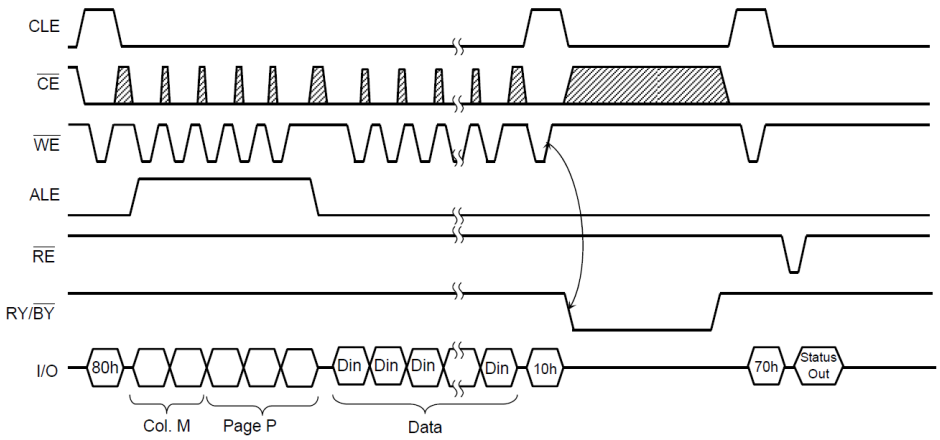

Figure 19. Page Program Timing Diagram

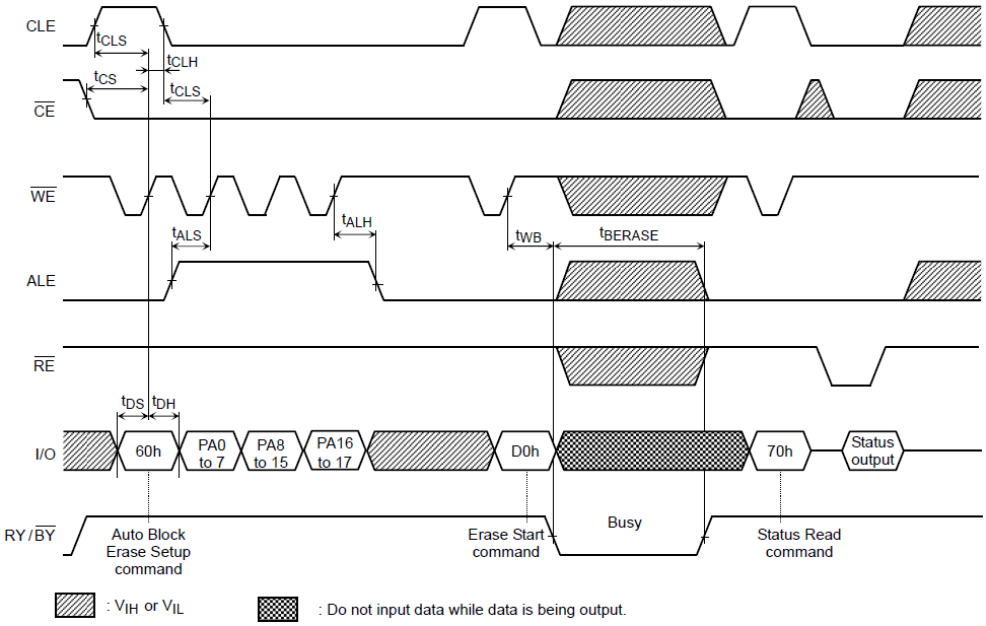

Figure 20. Block Erase Timing Diagram

The driver is tested using the procedure in the test plan.

The firmware that uses the driver is the Flash Translation Layer. The FTL requires two main functions that hook into the Sector Server layer of the application note. These functions are Read_Sector and Write_Sector, functions that read and write one sector to and from memory. Both functions accept a 32-bit Logical Block Address, and pointer to a 512-byte scratch buffer. The Write_Sector method is described below:

1. Translate the LBA to a Physical Block Address using a translation function
2. If the PBA is already being used, push the PBA to a linked-list based stack
3. If the PBA is not being used, store the PBA into the translation map with the LBA as a key
4. Convert the PBA to a 5-byte array that is readable by the flash memory IC driver
5. Call Page Program of the driver with the address of the scratch buffer and 5-byte PBA as arguments
6. Check status of the page program, if good, return 0, if not, loop from step 1

The Read_Sector method follows:

1. The PBA is found in the translation map by using the LBA
2. If the PBA does not exist, the status is set for a fail
3. If the PBA exists, the PBA is converted to a 5-byte array for the driver
4. A Page Read function is called with the 5-byte address and address of the buffer as arguments
5. The status is set to a pass, and returned

When programming, a translate function is called to translate an LBA to a PBA. To do this, a 10-bit random number is generated. The number is used to create a PBA from an LBA by masking out the upper ten bits, and inserts the random number.

The last part of the FTL that has not been touched on is the garbage collection. When data is overwritten, the FTL changes the PBA for the given LBA, and stores the PBA in a stack. When the garbage collection function is triggered, the stale PBAs within the stack are popped and erased.

Chapter VIII. Integration and Test Results

After testing, constructing, and developing the system, the system tests for the proof of concept were performed. The system tests include inserting the USB input into the computer and determine if the operating system recognizes the device as a USB Mass Storage Device, formatting the SSD, and writing to and reading files from the drive. These were run on the proof of concept. The proof of concept passed the test in which the system is recognized by the operating system, but stalls upon formatting the drive.

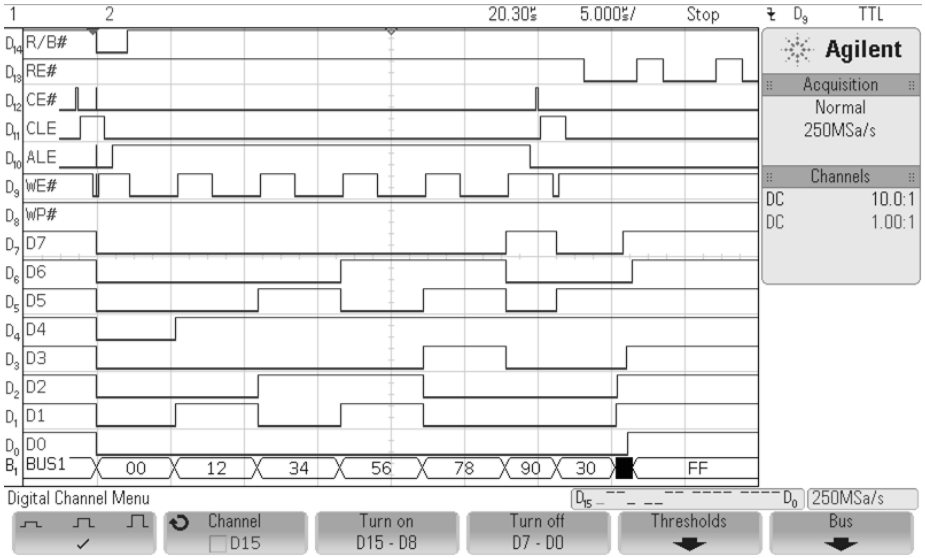

Figure 21. Page Read

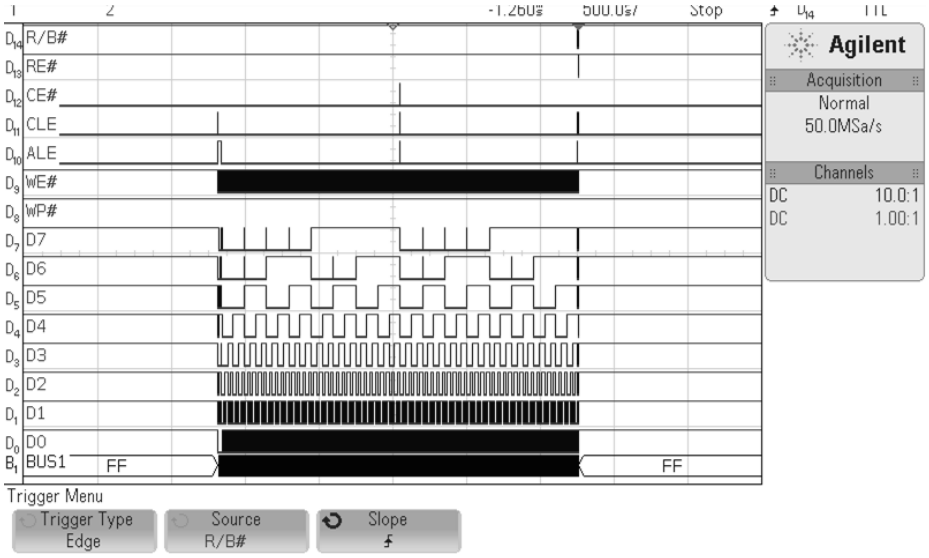

Figure 22. Page Program

Figures 21, 22, 23, and 24 all display the driver programming and reading the memory IC, in this case the TC58NVG3S0FTA00. The page read and page program routines in Figures 21 and 22 match the routines displayed in the datasheet shown in Figure 18 and 19. Figure 23 and 24 display a more in depth look at the program routine, as the MCU sends a command and address like the page read, but also waits for a status from the memory IC upon the end of a write.

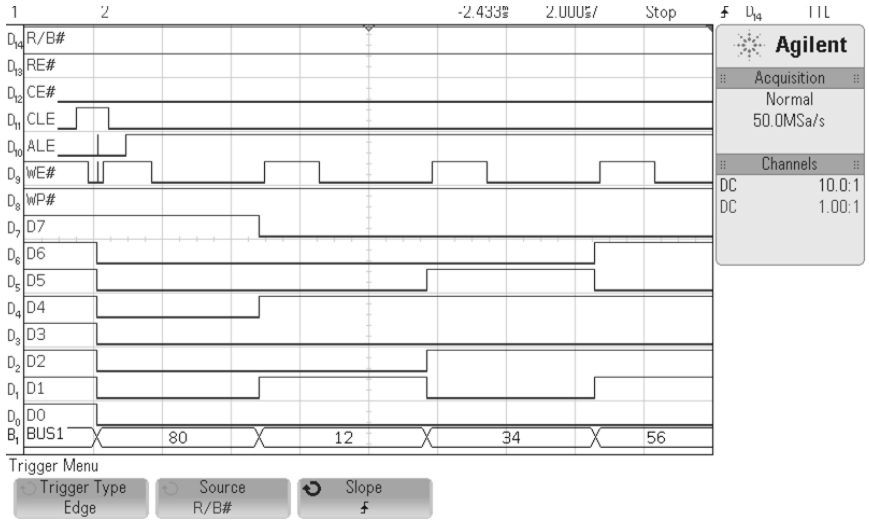

Figure 23. Page Program Command

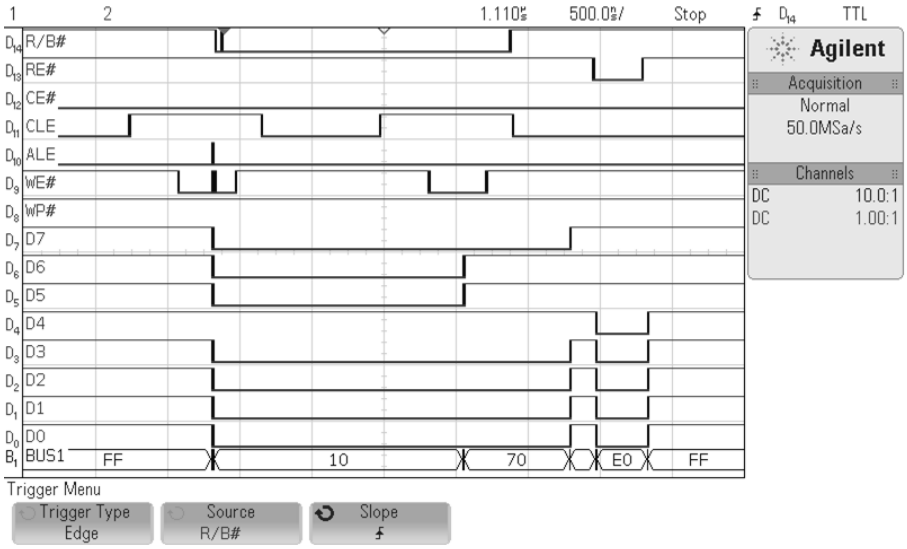

Figure 24. Page Program Status

Chapter IX. Conclusion

The project at the time of this report reached the proof of concept stage. The proof of concept is recognized by the operating system as a mass storage device, but stalls when trying to format, read, and write data to the drive. Based on the definition of a solid-state drive, the proof of concept does not reach the title of an SSD, as it cannot read and write. Unfortunately, there was insufficient time to debug the drive and bring it to completion. This should be a focus of future work.

Once the design is fully debugged and running, there are several improvements that could be made to improve the functionality and performance of the drive. These include:

1. Making the translation map more size efficient
2. Changing the translation algorithm from pseudo-random to choosing least recently used blocks
3. Using the Page Copy and Caching hardware functions of the memory IC to increase speed
4. Implementing multiple memory ICs
5. Implementing more RAM
6. Designing a custom PCB for the system
7. Updating the MCU for USB3.0+ compatibility

References

[1] Hruska, Joel. "Samsung Plans To Slash SSD Prices To Hard Drive Levels By 2020 - Extremetech". ExtremeTech. N.p., 2017. Web. 8 June 2017.

[2] "Advantages Of SSD Over HDD". Computerhope.com. N.p., 2017. Web. 8 June 2017.

[3] "Flash (SSD) Technology (And Beyond) Fundamentals". So-Cal Engineer. N.p., 2017. Web. 8 June 2017.

[4] Kareh, Badih el-. Silicon Devices And Process Integration. New York: Springer, 2009. Print.

[5] "Cylinder-Head-Sector". En.wikipedia.org. N.p., 2017. Web. 8 June 2017.

[6] USB Mass Storage Device Reference Design Programmer's Guide. Silicon Labs, 2017. Web. 8 June 2017.

[7] "Keil 8051 Microcontroller Development Tools". Keil.com. N.p., 2017. Web. 17 June 2017.

[8] Toshiba MOS Digital Integrated Circuit Silicon Gate CMOS 8 Gbit (1G X 8 Bit) CMOS NAND E2PROM. Toshiba, 2017. Web. 10 June 2017.

[9] E. Williams, "Environmental Impacts of Microchip Manufacture," Thin Solid Films, vol. 2, no. 049, Aug. 2004.

[10] "IEEE IEEE Code Of Ethics". Ieee.org. N.p., 2017. Web. 11 June 2017.

Appendix A

Table 4. ABET Senior Project Analysis

Project Title: Solid State Drive

Student's Name: Shaun Steele **Student's Signature: Shaun Steele**

Advisor's Name: Danowitz **Advisor's Initials:** **Date:** June 14, 2017

• 1. Summary of Functional Requirements
The system is a functioning external solid-state drive. It shall utilize USB protocol for communication and power. The storage capacity of 16GB or greater. System consists of at least an MCU and memory integrated circuit (IC). Memory IC is non-volatile NAND flash memory.

• 2. Primary Constraints
The constraints are listed in Chapter III. These constraints describe a minimum system function for a SSD within the scope of this project.

• 3. Economic
This project gives another option for people wanting long-term storage devices for their computers. It also provides Cal Poly a chance to prove its curriculum achieves sufficient ABET accreditation criteria. Every module and component used in the project requires design, testing, packaging, and shipping, which all requires paid workers. The components use processed resources and require transportation, both of which pollute surrounding environments [9]. If the outcome of the project is mass-produced, many consumers may purchase the SSD.

Costs accrue throughout the project, in the forms of my time spent working, and purchasing components after design completion. The benefits this project accrues throughout is the knowledge of an SSD's ins and outs, and producing a functioning SSD at the end. Cal Poly is also provided with a senior project.

The actual SSD requires a minimum of power and data inputs. This power and data comes from a computer. The total estimated cost of the project is $7,147.84 containing both labor and material costs. This leaves $465.62 for the estimated component costs. The real cost of components is $177.56, including test components cost. The entire project cost is $6,222.56 including labor costs of one engineer at $39/hour wage.

The hardware and software designs are completed on an already purchased computer and in Cal Poly EE labs. The labs also have testing materials for the SSD: power supply, digital multimeter, oscilloscope, and logic analyzer.

This project does not profit, as it does not include marketing and mass production. Knowledge of how an SSD works is gained, the senior project is completed enabling graduation eligibility, and Cal Poly receives a completed senior project sufficient under ABET accreditation criteria.

The project is completed in spring quarter, 2017. The project exists until the materials within the SSD degrade. Bug fixes are issued when necessary after project completion.

• 4. If manufactured on a commercial basis:
Table IX shows estimated profits if the SSD is commercially produced. The values are estimated using a cost estimate analysis. Devices sold number estimated by pessimistic value of 100, realistic value of 750, optimistic value of 1000. The annual profit is calculated using the equation and calculation below. N_{DS} represents Number of Devices Sold annually, P_D represents Purchase Price per Device, C_D represents Manufacturing Cost per Device, and C_L represents Total Labor Cost. The manufacturing cost per device is calculated using the price of the MCU ($8.12), memory ICs ($38.00), and an estimate for the Printed Circuit Board ($10), with a total of $56.12.

$$Annual\ Profit = N_{DS}(P_D - C_D) - C_L = 684(\$100 - \$56.12) - \$6{,}045 = \$23{,}968.92$$

Table 5. Numbers For Commercial Manufacturing

Commercial Variable	Amount
Number of Devices Sold Annually	683
Manufacturing Cost per Device	$56.12
Purchase Price per Device	$100
Annual Profit	$23,968.92
Cost per Unit Time	$44,431.08 per 10 years

• 5. Environmental

The environmental impacts for this project's materials stem from manufacturing of plastics and silicon, and the harmful emissions those processes create [9]. The use of an SSD affects the environment by using power typically supplied by a computer. The computer receives charge from the wall outlet, which receives power from a local power facility that has some detrimental effects on the surrounding environment.

Resources directly affected by SSDs are silicon, copper, and some materials used in plastics. The fabrication facilities and routes used for shipping indirectly affect the surrounding ecosystems. The pollution and erosion of facilities used to produce and ship materials for this project make a negative impact on the surrounding environments' air quality and habitat.

• 6. Manufacturability

Issues that may arise with manufacturing may stem from a lack of PCB design experience. Once experience is gained, the SSD may be able to be manufactured on a larger scale.

• 7. Sustainability

The challenges associated with maintaining the device rely on device and computer power while using the SSD. The project influences resources' sustainable use. This project proves the resources being used are in demand, giving fabricators and distributors a reason to supply such refined resources. This can negatively affect the fabricators' surrounding environment and the total supply of the resource itself.

A design upgrade based on sustainability relies on using recycled parts or materials with minimal impact on the environment to produce. This upgrade may hinder the SSD's performance due to an increase overall system costs.

• 8. Ethical

The decisions made for this project do not intend to harm or hinder any externalities surrounding the SSD realm. Precautions are taken to ensure a safely working product through testing by use. The environments surrounding fabricators feel the only negative effects of the project.

The estimates and claims remain honest, and honestly seem realistic to fit in the project's scope. Unjust compensation does not occur in this project.

This project teaches about SSDs and general computer hardware knowledge that is applicable to future projects that have a larger scope and help a larger population. This project's completion proves maintenance and improvement of my technological competence through learning, troubleshooting, and communication with others that have a higher competence.

Writing a project plan that peers regularly analyze and critique, and designs going under reviews displays the sixth tenant of the IEEE Code of Ethics [10]. Any constructive insight on the project is welcome. All potential contracts and interactions remain civil, professional, and maintained with positive action.

Psychological Egoism represents another ethical framework used to evaluate the ethics of the project. The fact that this senior project is necessary to graduate with a BSEE and having a functioning SSD with the knowledge behind how it works creates a self-interest in the project.

• **9. Health and Safety**
The processes that fabricators use make fumes that can harm the employees of the company. Working on the project can influence sleep and stress levels, so creation of a project plan and staying on top of the project allows for a successful project with manageable levels of stress.

• **10. Social and Political**
The largest political issue with this project stems from deciding to use a manufacturer that may outsource its fabrication due to decreased labor cost. If the price margin between a domestic or international distributor remains too large, I must use an international distributor to reach the cost specification.

This project affects Cal Poly's EE department and the companies the materials are purchased from. Those involved in fabrication and resource allocation remain indirect stakeholders. This project benefits many stakeholders by providing a complete project that qualifies ABET criteria and monetary exchanges with project material distributors. Fabricators harm the surrounding environment with pollution, thus harming any stakeholders within that environment.

Inequalities may arise in the location companies fabricate materials. Two different locations may make the same materials with the same quality and in the same timeframe, but one location may pay its employees less, creating an inequality of pay.

• **11. Development**
The project is successfully completed based on the understanding of Floating Gate Transistor technology, USB protocol and its interaction with file systems. Most of these topics do not appear in the Cal Poly EE curriculum, requiring independent study through a literature search. References for these studies remain on page 21.

The challenges faced with my project fall under learning how an SSD works, and learning the tools necessary to implement a successful SSD. Comfortability of researching and applying what was learned into a firmware level program is necessary for this project's completion.

Appendix B

Table V. Bill of Materials

No.	Name	Description	Distributor	Qty.	Price	Extended Price
1	Silicon Labs C8051F340-GQ	8051 C8051F34x Microcontroller IC, 8-Bit, 48MHz, 64KB Flash, 48-QFP	Digikey	1	$8.12	$8.12
2	Toshiba Semiconductor and Storage TC58NVG3S0FTA00	Flash – NAND Memory IC, 8Gb, Parallel, 25ns, 48-TSOP	Digikey	1	$10.46	$10.46
3	Silicon Labs TOOLSTICK342MPP	C8051F34X Toolstick Microcontroller Programmer	Digikey	1	$92.24	$92.24
4	TSOP48 to DIP48 Socket Adapter	TSOP48 to DIP48 Socket Adapter	AliExpress	1	$6.50	$6.50
5	Cypress Semiconductor Corp FM25W256-G	FRAM (Ferroelectric RAM) Memory IC, 32KB, SPI, 8-SOIC	Digikey	1	$5.81	$5.81
6	Micron Technology Inc. MT29F64G08CBABAWP:B TR	Flash-NAND Memory IC, 64Gb, Parallel, 48-TSOP	Digikey	4	$7.75	$31.00
7	Logical Systems Inc. PA-SOD3TK-08/5	8-SOIC to 8-DIP Adapter	Digikey	1	$20.00	$20.00
Total						$174.13

Appendix C

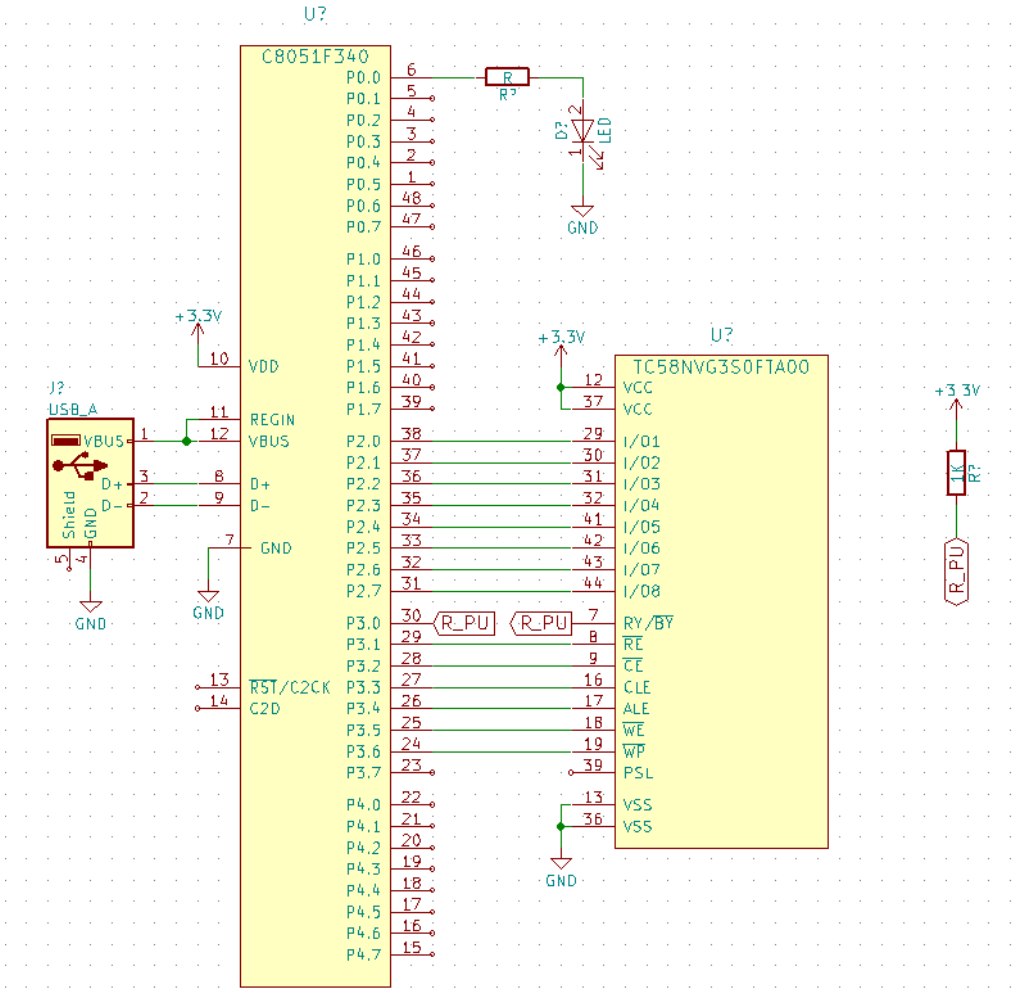

Figure 25. Proof of Concept Schematic

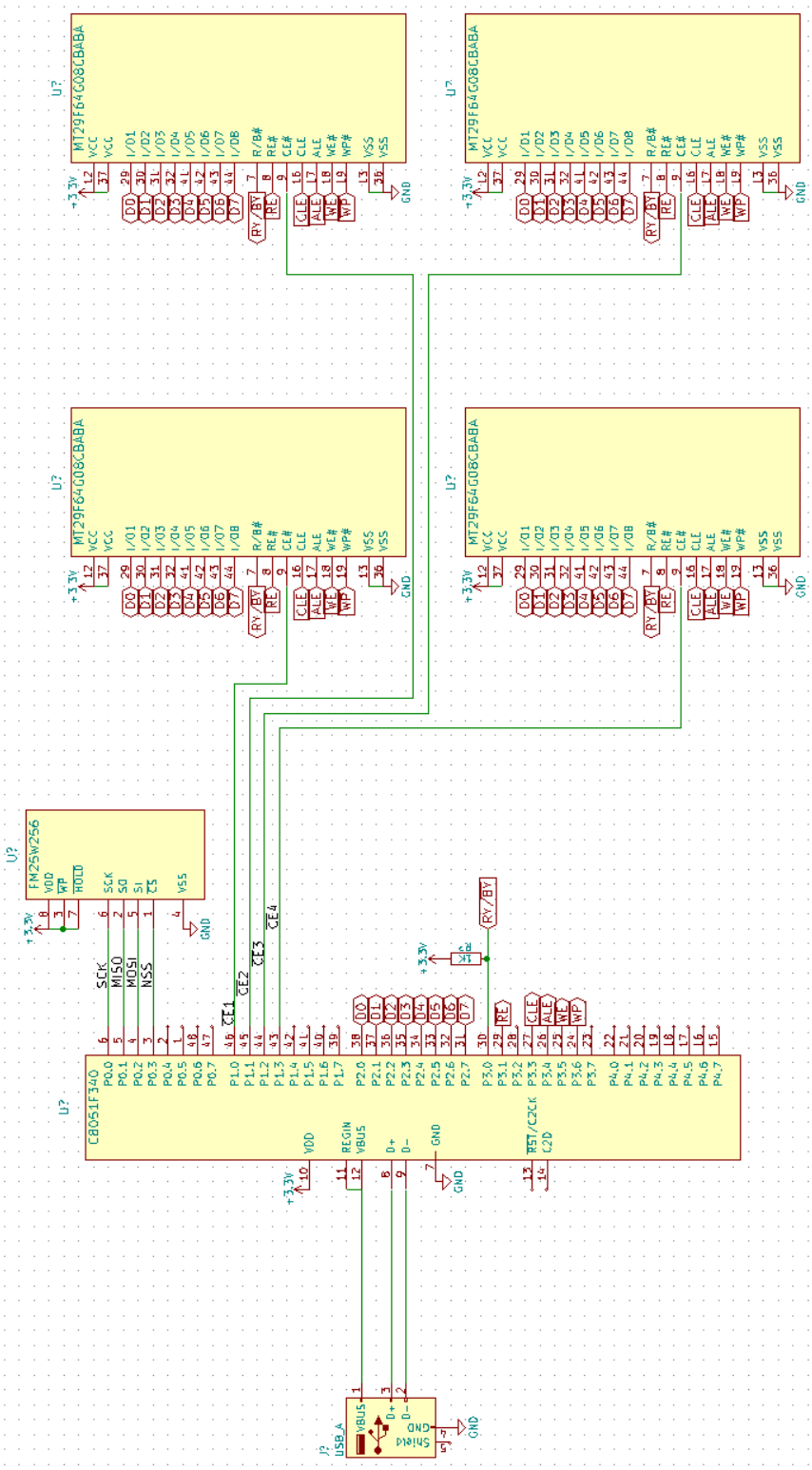

Figure 26. 32GB SSD Schematic

Appendix D

Figure 27. Main Function Files

```
//-----------------------------------------------------------------------------
// F34x_MSD_USB_Main.c
//-----------------------------------------------------------------------------
// Copyright 2009 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// File contains the main loop of application. It also contains small delay
// function and some simple initializations (CPU, Ports)
//
// How To Test:    See Readme.txt
//
//
// FID:            34X000062
// Target:         C8051F34x
// Tool chain:     Keil
// Command Line:   See Readme.txt
// Project Name:   F34x_USB_MSD
//
// REVISIONS:  11/22/02 - DM:  Added support for switches and sample USB
// interrupt application.
//
// Release 1.3
//    -All changes by ES
//    -18 DEC 2009
//    -No changes; incremented revision number to match project revision
//
// Release 1.2
//    -All changes by BD and PD
//    -5 OCT 2007
//    -No changes; incremented revision number to match project revision
//
// Release 1.1
//    -All changes by PKC
//    -09 JUN 2006
//    -Replaced SFR definitions file "c8051f320.h" with "c8051f340.h"
//
// Release 1.0
//    -Initial Release
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include "F34x_MSD_Definitions.h"
#include "F340_MSD_Physical_Settings.h"
//#include "F34x_MSD_CF_Basic_Functions.h"
#include "c8051f340.h"
#include "F34x_MSD_USB_Register.h"
#include "F34x_MSD_USB_Main.h"
#include "F34x_MSD_USB_Descriptor.h"
//#include "F34x_MSD_VBUS_Functions.h"

#include "F34x_MSD_Sect_Serv.h"
#include "F34x_MSD_Msd.h"
//#include "F34x_MSD_UART.h"
#include "F34x_MSD_File_System.h"

#include "F340_MSD_FTL.h"
#include "F340_MSD_Flash_Basic_Functions.h"
#include <stdio.h>


//#include "F34x_MSD_Cmd.h" //wozb - 27-09-2005 - it's not need it was used to communicate via uart
//#include "F34x_MSD_Log.h" //wozb - 27-09-2005 - it's not need it was used to communicate via uart

//#include "F34x_MSD_Temp_Sensor.h"

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F34x
//-----------------------------------------------------------------------------

sfr16 TMR2RL   = 0xca;                  // Timer2 reload value
```

```
sfr16 TMR2     = 0xcc;                  // Timer2 counter

sbit C_PWR = P1^7;
//-----------------------------------------------------------------------------
// Main Routine
//-----------------------------------------------------------------------------
volatile DWORD xdata tickcount=0;

//extern void Wait_ms(unsigned int count);

//-----------------------------------------------------------------------------
// main
//-----------------------------------------------------------------------------
//
// Main loop
//
// Parameters   :
// Return Value :
//-----------------------------------------------------------------------------

void main(void) {
        PCA0MD &= ~0x40;                        // Disable Watchdog timer

        Sys_Clk_Init();                          // Initialize oscillator

        REG0CN &= ~(0xC0);                                                      // Initialize Voltage
Reference
        REG0CN |= 0x80;

        Port_Init();                            // Initialize crossbar and GPIO

        Init_Flash();                                                   // Initialize Flash Driver

        FTL_Init();                                                             // Initialize Flash
Translation Layer

        //UART0_Init();

//#ifdef __F340_VER__
//      Init_Temp_Sensor();                                             // initialize temerature sensor
//#endif
        Sect_Init();

        Timer_Init();                           // Initialize timer2

        USB0_Init();                            // Initialize USB0



//#ifdef __F340_VER__
        FileSys_Init();
//#endif

//      Cmd_Init();



        USB_Bulk_Init();


        while (1) {
//              Cmd_Step();

                Msd_Step();

/*              Log_Step();
#ifdef __F340_VER__
                Temp_Log_Step();
                Switch_On_Off_UART();
#endif
*/
        }

}

//-----------------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Sys_Clk_Init
```

```
//-----------------------------------------------------------------------------
//
// This routine initilizes the system clock and USB clock
//
// Parameters   :
// Return Value :
//-----------------------------------------------------------------------------

void Sys_Clk_Init(void)
{
#ifdef _USB_LOW_SPEED_

   OSCICN |= 0x03;                      // Configure internal oscillator for
                                        // its maximum frequency and enable
                                        // missing clock detector

   CLKSEL  = SYS_INT_OSC;               // Select System clock
   CLKSEL |= USB_INT_OSC_DIV_2;         // Select USB clock
#else
   OSCICN |= 0x03;                      // Configure internal oscillator for
                                        // its maximum frequency and enable
                                        // missing clock detector

   CLKMUL  = 0x00;                      // Select internal oscillator as
                                        // input to clock multiplier

   CLKMUL |= 0x80;                      // Enable clock multiplier
   Delay();                             // Delay for clock multiplier to begin
   CLKMUL |= 0xC0;                      // Initialize the clock multiplier
   Delay();

   while(!(CLKMUL & 0x20));               // Wait for multiplier to lock
#ifdef __F340_VER__
#ifdef F340_24M
  CLKSEL = SYS_4X_DIV_2;
#else
  CLKSEL = SYS_4X_MUL;
#endif
#else
#ifdef __F326_VER__
   CLKSEL  = SYS_4X_DIV_2;
#else
   CLKSEL  = SYS_INT_OSC;               // Select system clock
#endif
#endif
   CLKSEL |= USB_4X_CLOCK;              // Select USB clock
#endif  /* _USB_LOW_SPEED_ */
}


//-----------------------------------------------------------------------------
// Port_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
//
// Parameters   :
// Return Value :
//-----------------------------------------------------------------------------

void Port_Init(void) {
        // Default values on reset:
        // P0MDIN=0xFF;           P1MDIN=0xFF;     P2MDIN=0xFF;     P3MDIN=0xFF;
        // P0MDOUT=0x00; P1MDOUT=0x00;    P2MDOUT=0x00;    P3MDOUT=0x00;
        // P0SKIP=0x00;           P1SKIP=0x00;     P2SKIP=0x00;
        // XBR0=0x00;
        //data   int i;
/*#ifdef __F326_VER__

        GPIOCN |= 0x40;
        P0MDOUT = 0x0d;
        P0 |= 0x02;
        SCS = 1;
        SCLK = 1;
#else*/
        P0MDOUT |= 0xFF;                  // Initialize P0 as PushPull
        P2MDOUT |= 0xFF;                  // Initialize P2 as PushPull
        P3MDOUT |= 0x7F;                  // Initialize P3.6 downto P3.0 as PushPull
        P0 &= ~(0x01);
        XBR1 |= 0x40;                             // Set X-Bar
```

```
/*       P1MDIN = 0xff;

//   P1MDIN  = 0x7F;                        // Port 1 pin 7 set as analog input
   ////p1.0 - out,p1.1-out,p1.2 - out,p1.3 -in,p1.4 - in p1.6 - out
   P1MDOUT = 0xC7;
   P1 = 0x18;
   P3MDIN = 0xff;
   P3MDOUT = 0xFF;
   P3 &= ~(0xe0) ;
   P4MDIN = 0xff;
   P4MDOUT = 0x00;
   P4  = 0xff;
   P0MDOUT = 0x1D;                    // enable TX0,SCK,MOSI as a push-pull
   P2MDOUT = 0x0C;                    // enable LEDs as a push-pull output
   XBR0    = 0x03;                     // UART0 TX and RX pins enabled, SPI enabled
   XBR1=(0x40 | 0x80); // Enable crossbar, disable weak pull-up
  // XBR1=0x40 ;
   CF_OE = 0;
   C_PWR = 1;
         for(i=0;i<5000;i++)
            Delay();
   XBR1 &= ~ 0x80;
   CF_OE = 1;
   C_PWR = 0;
         for(i=0;i<5000;i++)
            Delay();
   XBR1 |= 0x80; // week pull-up off for CF*/
//#endif

}


//-----------------------------------------------------------------------------
// USB0_Init
//-----------------------------------------------------------------------------
//
// USB Initialization
// - Initialize USB0
// - Enable USB0 interrupts
// - Enable USB0 transceiver
// - Enable USB0 with suspend detection
//
// Parameters   :
// Return Value :
//-----------------------------------------------------------------------------

void USB0_Init(void)
{
   POLL_WRITE_BYTE(POWER,  0x08);          // Force Asynchronous USB Reset
   POLL_WRITE_BYTE(IN1IE,  0x07);          // Enable Endpoint 0-2 in interrupts
   POLL_WRITE_BYTE(OUT1IE, 0x07);          // Enable Endpoint 0-2 out interrupts
   POLL_WRITE_BYTE(CMIE,   0x07);          // Enable Reset, Resume, and Suspend interrupts
/*#ifdef _USB_LOW_SPEED_
   USB0XCN = 0xC0;                         // Enable transceiver; select low speed
   POLL_WRITE_BYTE(CLKREC, 0xA0);          // Enable clock recovery; single-step mode
                                           // disabled; low speed mode enabled
*/
//#else
   USB0XCN = 0xE0;                         // Enable transceiver; select full speed
   POLL_WRITE_BYTE(CLKREC, 0x80);          // Enable clock recovery, single-step mode
                                           // disabled
//#endif /* _USB_LOW_SPEED_ */
   if (REG0CN & 0x40) {
            P0 |= 0x01;                                                  // Set LED for VBUS Active
   }

   EIE1 |= 0x02;                           // Enable USB0 Interrupts
   EA = 1;                                 // Global Interrupt enable
                                           // Enable USB0 by clearing the USB Inhibit bit
   POLL_WRITE_BYTE(POWER,  0x01);          // and enable suspend detection


}


//-----------------------------------------------------------------------------
// Timer_Init
//-----------------------------------------------------------------------------
//
// Timer initialization
// - Timer 2 reload, used to check if switch pressed on overflow and
```

```
// used for ADC continuous conversion
//
// Parameters   :
// Return Value :
//-------------------------------------------------------------------------

void Timer_Init(void)
{
#ifdef __F326_VER__
        TMOD = 0x01;                                            //Timer 0 - 16 bit timer
        CKCON = 0x00;
        TL0             = 0x2e;
        TH0             = 0xf8;
        ET0             = 1;                                            // enable timer 0 interrupt
    TR0          = 1;                                           // start timer 0
#else

   TMR2CN  = 0x00;                      // Stop Timer2; Clear TF2;

   CKCON  &= ~0xF0;                     // Timer2 clocked based on T2XCLK;
   TMR2RL  = 0xF000;                    // Initialize reload value
   TMR2    = 0xffff;                    // Set to reload immediately

   ET2     = 1;                         // Enable Timer2 interrupts
   TR2     = 1;                         // Start Timer2
#endif
}


//-------------------------------------------------------------------------
// Timer2_ISR
//-------------------------------------------------------------------------
//
// Timer 2 interrupt routine
// Called when timer 2 overflows
//
// Parameters   :
// Return Value :
//-------------------------------------------------------------------------

//#ifndef __F326_VER__
void Timer2_ISR(void) interrupt 5 {
        /*
        tickcount++;
#ifdef __F340_VER__
#ifdef F340_24M
        TMR2RL = 0xF82e;
#endif
        TMR2RL = 0xF05e;
//      TMR2RL = 0xFC16;
#else
        TMR2RL = 0xFC16;          // Re-initialize reload value (1kHz, 1ms)
#endif
        //TMR2RL = 0xE0B0;                // Re-initialize reload value (125Hz, 8ms)
        */
        Collect_Garbage();
        TF2H=0;                         // Clear interrupt
}

/*#else

//-------------------------------------------------------------------------
// Timer0_ISR
//-------------------------------------------------------------------------
//
// Timer 0 interrupt routine
// Called when timer 0 overflows
//
// Parameters   :
// Return Value :
//-------------------------------------------------------------------------
void Timer0_ISR(void) interrupt 1
{

        TL0             += 0x2e;
        TH0             = 0xf8;
        tickcount++;
}*/
//#endif
```

```
//-----------------------------------------------------------------------------
// Delay
//-----------------------------------------------------------------------------
//
// Used for a small pause, approximately 80 us in Full Speed,
// and 1 ms when clock is configured for Low Speed
//
// Parameters   :
// Return Value :
//-----------------------------------------------------------------------------

void Delay(void)
{
   data int x;
   for(x = 0;x < 500;x)
      x++;
}
```

```
//-----------------------------------------------------------------------------
// F34x_MSD_USB_Main.h
//-----------------------------------------------------------------------------
// Copyright 2009 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// Main header file for USB firmware. Includes function prototypes,
// standard constants, and configuration constants.//
//
// FID:            34X000063
// Target:         C8051F34x
// Tool chain:     Keil
// Command Line:   See Readme.txt
// Project Name:   F34x_USB_MSD
//
// Release 1.3
//    -All changes by ES
//    -18 DEC 2009
//    -No changes; incremented revision number to match project revision
//
// Release 1.2
//    -All changes by BD and PD
//    -5 OCT 2007
//    -No changes; incremented revision number to match project revision
//
// Release 1.1
//    -All changes by PKC
//    -09 JUN 2006
//    -No changes; incremented revision number to match project revision
//
// Release 1.0
//    -Initial Release
//
// 11/22/02 - DM: 1. Updated function prototypes and added constants
//                to F34x_USB_Main.h with sample interrupt firmware.

//-----------------------------------------------------------------------------
// Header File Preprocessor Directive
//-----------------------------------------------------------------------------

#ifndef _USB_MAIN_H_
#define _USB_MAIN_H_

#include "F34x_MSD_Definitions.h"
//#define _USB_LOW_SPEED_                      // Change this comment to make Full/Low speed

#define SYSCLK                   12000000    // SYSCLK frequency in Hz

// USB clock selections (SFR CLKSEL)
#define USB_4X_CLOCK             0x00        // Select 4x clock multiplier, for USB Full Speed
#define USB_INT_OSC_DIV_2        0x10        // See Data Sheet section 13. Oscillators
#define USB_EXT_OSC              0x20
#define USB_EXT_OSC_DIV_2        0x30
#define USB_EXT_OSC_DIV_3        0x40
#define USB_EXT_OSC_DIV_4        0x50

// System clock selections (SFR CLKSEL)
#define SYS_INT_OSC              0x00        // Select to use internal oscillator
#define SYS_4X_MUL               0x03        // Select to use internal oscillator
#define SYS_EXT_OSC              0x01        // Select to use an external oscillator
#define SYS_4X_DIV_2             0x02
```

```
// BYTE type definition
#ifndef _BYTE_DEF_
#define _BYTE_DEF_
typedef unsigned char BYTE;
#endif   /* _BYTE_DEF_ */

// WORD type definition, for KEIL Compiler
#ifndef _WORD_DEF_                          // Compiler Specific, written for Little Endian
#define _WORD_DEF_
typedef union {unsigned int i; unsigned char c[2];} WORD;
#define LSB 1                               // All words sent to and received from the host are
#define MSB 0                               // little endian, this is switched by software when
                                            // neccessary.  These sections of code have been marked
                                                                                        // with
"Compiler Specific" as above for easier modification
#endif   /* _WORD_DEF_ */

// DWORD type definition
#ifndef _DWORD_DEF_
#define _DWORD_DEF_
typedef unsigned long DWORD;
#endif /* _DWORD_DEF_ */

extern volatile DWORD xdata tickcount;

// Define Endpoint Packet Sizes
#ifdef _USB_LOW_SPEED_
#define  EP0_PACKET_SIZE         0x40        // This value can be 8,16,32,64 depending on device speed, see USB
spec
#else
#define  EP0_PACKET_SIZE         0x40
#endif /* _USB_LOW_SPEED_ */

#define  EP1_PACKET_SIZE         0x0040      // Can range 0 - 1024 depending on data and transfer type
#define  EP1_PACKET_SIZE_LE      0x4000      // IMPORTANT- this should be Little-Endian version of
EP1_PACKET_SIZE
#define  EP2_PACKET_SIZE         0x0040      // Can range 0 - 1024 depending on data and transfer type
#define  EP2_PACKET_SIZE_LE      0x4000      // IMPORTANT- this should be Little-Endian version of
EP2_PACKET_SIZE

// Standard Descriptor Types
#define  DSC_DEVICE              0x01        // Device Descriptor
#define  DSC_CONFIG              0x02        // Configuration Descriptor
#define  DSC_STRING              0x03        // String Descriptor
#define  DSC_INTERFACE           0x04        // Interface Descriptor
#define  DSC_ENDPOINT            0x05        // Endpoint Descriptor

// HID Descriptor Types
#define DSC_HID                                 0x21             // HID Class Descriptor
#define DSC_HID_REPORT                    0x22             // HID Report Descriptor

// Standard Request Codes
#define  GET_STATUS              0x00        // Code for Get Status
#define  CLEAR_FEATURE           0x01        // Code for Clear Feature
#define  SET_FEATURE             0x03        // Code for Set Feature
#define  SET_ADDRESS             0x05        // Code for Set Address
#define  GET_DESCRIPTOR          0x06        // Code for Get Descriptor
#define  SET_DESCRIPTOR          0x07        // Code for Set Descriptor(not used)
#define  GET_CONFIGURATION       0x08        // Code for Get Configuration
#define  SET_CONFIGURATION       0x09        // Code for Set Configuration
#define  GET_INTERFACE           0x0A        // Code for Get Interface
#define  SET_INTERFACE           0x0B        // Code for Set Interface
#define  SYNCH_FRAME             0x0C        // Code for Synch Frame(not used)
#define  MSD_RESET                           0xFF             // Mass-storage device Reset
#define  MSD_GET_MAX_LUN          0xFE       // Mass-storage device Get Max LUN

// HID Request Codes
#define GET_REPORT                              0x01             // Code for Get Report
#define GET_IDLE                          0x02             // Code for Get Idle
#define GET_PROTOCOL                      0x03             // Code for Get Protocol
#define SET_REPORT                              0x09             // Code for Set Report
#define SET_IDLE                          0x0A             // Code for Set Idle
#define SET_PROTOCOL                      0x0B             // Code for Set Protocol

// Define device states
#define  DEV_ATTACHED            0x00        // Device is in Attached State
#define  DEV_POWERED             0x01        // Device is in Powered State
#define  DEV_DEFAULT             0x02        // Device is in Default State
#define  DEV_ADDRESS             0x03        // Device is in Addressed State
#define  DEV_CONFIGURED          0x04        // Device is in Configured State
```

```
#define  DEV_SUSPENDED           0x05        // Device is in Suspended State

// Define bmRequestType bitmaps
#define  IN_DEVICE               0x00        // Request made to device, direction is IN
#define  OUT_DEVICE              0x80        // Request made to device, direction is OUT
#define  IN_INTERFACE            0x01        // Request made to interface, direction is IN
#define  OUT_INTERFACE           0x81        // Request made to interface, direction is OUT
#define  IN_ENDPOINT             0x02        // Request made to endpoint, direction is IN
#define  OUT_ENDPOINT            0x82        // Request made to endpoint, direction is OUT

// Define wIndex bitmaps
#define  IN_EP1                  0x81        // Index values used by Set and Clear feature
#define  OUT_EP1                 0x01        // commands for Endpoint_Halt
#define  IN_EP2                  0x82
#ifdef __F326_VER__
#define  OUT_EP2                            0x01
#else
#define  OUT_EP2                            0x01
#endif

// Define wValue bitmaps for Standard Feature Selectors
#define  DEVICE_REMOTE_WAKEUP    0x01        // Remote wakeup feature(not used)
#define  ENDPOINT_HALT           0x00        // Endpoint_Halt feature selector

// Define Endpoint States
#define  EP_IDLE                 0x00        // This signifies Endpoint Idle State
#define  EP_TX                   0x01        // Endpoint Transmit State
#define  EP_RX                   0x02        // Endpoint Receive State
#define  EP_HALT                 0x03        // Endpoint Halt State (return stalls)
#define  EP_STALL                0x04        // Endpoint Stall (send procedural stall next status phase)
#define  EP_ADDRESS              0x05        // Endpoint Address (change FADDR during next status phase)

// Function prototypes
// USB Routines
void USB_Resume(void);                       // This routine resumes USB operation
void USB_Reset(void);                        // Called after USB bus reset
void Handle_Setup(void);                     // Handle setup packet on Endpoint 0
void Handle_In1(BYTE* ptr_buf);                      // Handle in packet on Endpoint 1
void Handle_Out2(void);                      // Handle out packet on Endpoint 2
void USB_Suspend(void);                      // This routine called when suspend signalling on bus
void Out2_Get_Data(BYTE* ptr_buf);                           // Copies from FIFO to ptr_buf, clears FIFO-full
flag.
void Out2_Done(void);                                                // Call this when finished with the data

// Standard Requests
void Get_Status(void);                       // These are called for each specific standard request
void Clear_Feature(void);
void Set_Feature(void);
void Set_Address(void);
void Get_Descriptor(void);
void Get_Configuration(void);
void Set_Configuration(void);
void Get_Interface(void);
void Set_Interface(void);

// MSD Specific Requests
// void Reset_Msd(void);
// void Get_MaxLUN(void);

// Initialization Routines
void Sys_Clk_Init(void);                      // Initialize the system clock(depends on Full/Low speed)
void Port_Init(void);                        // Configure ports for this specific application
void USB0_Init(void);                        // Configure USB core for either Full/Low speed
void Timer_Init(void);                       // Start timer 2 for use by ADC and to check switches
void Adc_Init(void);                         // Configure ADC for continuous conversion, low-power mode

// Other Routines
void Timer2_ISR(void);                       // Called when Timer 2 overflows, see if switches are pressed
void Adc_ConvComple_ISR(void);               // When a conversion completes, switch ADC multiplexor
void USB_ISR(void);                          // Called to determine type of USB interrupt
void Fifo_ReadC(BYTE, unsigned int, BYTE *) ;
extern void Fifo_Read (BYTE, unsigned int, BYTE *) ; // Used for multiple byte reads of Endpoint fifos
extern void Fifo_Write (BYTE, unsigned int, BYTE *) reentrant;// Used for multiple byte writes of Endpoint
fifos
void Force_Stall(void);                      // Forces a procedural stall on Endpoint 0
void Delay(void);                            // Approximately 80 us/1 ms on Full/Low Speed

void USB_In(BYTE* ptr_buf,unsigned count);

void USB_Bulk_Init(void);
```

```
extern unsigned xdata Out_Count;
extern BYTE xdata Out_Packet[EP2_PACKET_SIZE];
extern BYTE xdata In_count;
extern BYTE xdata In_Packet[EP1_PACKET_SIZE];
//extern BYTE xdata In_Overrun;

#endif      /* _USB_MAIN_H_ */
```

Figure 28. Memory IC Driver

```
//-------------------------------------------------------------------------
// F34x_MSD_Flash_Basic_Functions.c
//-------------------------------------------------------------------------
#include "F340_MSD_Physical_Settings.h"
#include "F340_MSD_Flash_Basic_Functions.h"
#include <stdint.h>
#include <stdio.h>

//-------------------------------------------------------------------------
// Init_Flash
//-------------------------------------------------------------------------
//
// Initializes flash memory from boot
//
// Parameters   : void
// Return Value : void
//-------------------------------------------------------------------------
void Init_Flash(void) {
        Power_On();
        Reset();
}

//-------------------------------------------------------------------------
// Power_On
//-------------------------------------------------------------------------
//
// Boot Sequence
//
// Parameters   : void
// Return Value : void
//-------------------------------------------------------------------------
void Power_On(void) {
        P3 &= ~(0x18);
        WP = 1;
        FLASH_WAIT_BUSY;
}

//-------------------------------------------------------------------------
// ID_Read
//-------------------------------------------------------------------------
//
// Gets ID of Flash memory
//
// Toshiba TH58NVG3S0HTA00 ID
//  ID[0] = 0x98
//  ID[1] = 0xD3
//  ID[2] = 0x90
//  ID[3] = 0x26
//  ID[4] = 0x76
//
// Parameters   : char
// Return Value : char
//-------------------------------------------------------------------------
void ID_Read(uint8_t * str) {
        int i;
        uint8_t addr[] = {0x00};
        FLASH_CMD(0x90);
        FLASH_ADDR_INPUT(addr,1);
        for(i=0; i<5; i++) {
                FLASH_DQ_LATCH_OUT(str[i]);
        }
        FLASH_WAIT_BUSY;
}

//-------------------------------------------------------------------------
// Page_Read
//-------------------------------------------------------------------------
//
// Reads values at address in flash memory
```

```
//
// Parameters   : char_ptr, char_ptr
// Return Value : void
//-----------------------------------------------------------------------------
void Page_Read(uint8_t * addr, uint8_t * buf) {
        int i;
        FLASH_CMD(0x00);
        FLASH_ADDR_INPUT(addr,5);
        FLASH_CMD(0x30);
        FLASH_WAIT_BUSY;
        for(i=0;i<PHYSICAL_BLOCK_SIZE;i++) {
                FLASH_DQ_LATCH_OUT(buf[i]);
        }
}

//-----------------------------------------------------------------------------
// Page_Program
//-----------------------------------------------------------------------------
//
// Programs values at address in flash memory
//
// Parameters   : char_ptr, char_ptr
// Return Value : char
//-----------------------------------------------------------------------------
uint8_t Page_Program(uint8_t * addr, uint8_t * buf) {
        int j;
        uint8_t status;

        FLASH_CMD(0x80);
                FLASH_ADDR_INPUT(addr,5);
                for(j=0;j<PHYSICAL_BLOCK_SIZE;j++) {
                        FLASH_DQ_LATCH_IN(buf[j]);
                }
                FLASH_CMD(0x10);
                CE = 1;
                FLASH_WAIT_BUSY;
                FLASH_STATUS_READ(status);
                return status;
}

//-----------------------------------------------------------------------------
// Block_Erase
//-----------------------------------------------------------------------------
//
// Erases block at block address
//
// Parameters   : char_ptr
// Return Value : char
//-----------------------------------------------------------------------------
uint8_t Block_Erase(uint8_t * addr) {
        uint8_t status;
        FLASH_CMD(0x60);
        FLASH_ADDR_INPUT(addr,2);
        FLASH_CMD(0xD0);
        FLASH_WAIT_BUSY;
        FLASH_STATUS_READ(status);
        return status;
}


//-----------------------------------------------------------------------------
// Reset
//-----------------------------------------------------------------------------
//
// Issues a reset command
//
// Parameters   : void
// Return Value : void
//-----------------------------------------------------------------------------
void Reset(void) {
        FLASH_CMD(0xFF);
        FLASH_WAIT_BUSY;
}
```

```
//-----------------------------------------------------------------------------
// F34x_MSD_Flash_Basic_Functions.h
//-----------------------------------------------------------------------------
#ifndef __FLASH_BASIC_H__
#define __FLASH_BASIC_H__

#include <C8051F340.h>
#include <intrins.h>
```

```
#include <stdio.h>
#include <stdint.h>

sbit WP =         P3^6;    // Active low
sbit WE =         P3^5;    // Active low
sbit ALE =        P3^4;    // Active high
sbit CLE =        P3^3;    // Active high
sbit CE =         P3^2;    // Active low
sbit RE =         P3^1;    // Active low
sbit RYBY =       P3^7;

//sbit DQ =       P2;


#define FLASH_CMD(value) {\
         P3 &= ~(0x14);\
         CE = 0;\
         CLE = 1;\
         FLASH_DQ_LATCH_IN(value);\
         CLE = 0;\
}\

#define FLASH_ADDR_INPUT(value,max) {\
         int i;\
         CE=0;\
         CLE=0; ALE=1;\
         for(i=0;i<max;i++) {\
                  FLASH_DQ_LATCH_IN(value[i]);\
         }\
         ALE = 0;\
}

// Read from flash
#define FLASH_DQ_LATCH_OUT(value) {\
         CE = 0;\
         P2MDOUT |= 0x00;\
         P2 |= 0xFF;\
         RE = 0;\
         value = P2;\
         RE = 1;\
}

// Write to flash
#define FLASH_DQ_LATCH_IN(value) {\
         CE = 0;\
         P2MDOUT &= 0xFF;\
         WE = 0;\
         P2 = value;\
         WE = 1;\
}\

#define FLASH_WAIT_BUSY {\
         while(!RYBY);\
}\

#define FLASH_STATUS_READ(value) {\
         FLASH_CMD(0x70);\
         FLASH_DQ_LATCH_OUT(value);\
}\

#define FLASH_IDLE {\
         CE = 1;\
}\

void Init_Flash(void);
void Page_Read(uint8_t * addr, uint8_t * buf);
uint8_t Page_Program(uint8_t * addr, uint8_t * buf);
uint8_t Block_Erase(uint8_t * addr);
void Page_Copy(uint8_t old_addr, uint8_t new_addr);
void Reset(void);
void ID_Read(uint8_t * str);
void Power_On(void);

#ifndef MACRO_FLASH_VERSIONS

#endif
#endif
```

Figure 29. Flash Translation Layer

```
//-----------------------------------------------------------------------------
// F34x_MSD_FTL.c
//-----------------------------------------------------------------------------
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "F34x_MSD_Definitions.h"
#include "F34x_MSD_Sect_Serv.h"
#include "F34x_MSD_Util.h"
#include "F340_MSD_Flash_Basic_Functions.h"
#include "F340_MSD_FTL.h"


struct Node {
    uint32_t val;
    struct Node *next;
} *top = NULL;

xdata uint16_t translation_map[LBA_PBA_SIZE]; // SAVE TO ON MCU FLASH


void FTL_Init(void) {
        memset(&(translation_map[0]),-1,LBA_PBA_SIZE*sizeof(translation_map[0]));
        srand(0);
}


uint8_t Read_Sector(uint32_t LBA, uint8_t* buffer) {
        uint8_t status, addr[5];
        uint32_t PBA;

        PBA = translation_map[LBA];
        if(PBA == -1) {
                status = -1;
        } else {
                ptoa(&(addr[0]),PBA);
                Page_Read(&(addr[0]),&buffer);
                status = 0; // Pass
        }

        return 0;
}


uint8_t Write_Sector(uint32_t LBA, uint8_t* buffer) {
        uint8_t addr[5], status = 0;
        uint32_t PBA;

        do {
                PBA = Translate(LBA);

                if(PBA != -1) {
                        push(PBA);
                } else {
                        translation_map[LBA] = PBA;
                }

                ptoa(&(addr[0]),PBA);

                status = Page_Program(&(addr[0]),&buffer);
        } while((status & 0x21) == 0x21);

        return 0;
}

uint32_t Translate(uint32_t LBA) {
        uint32_t PBA, shuf;

        shuf = MASK >> SHAMT; //0x0 - 0x1FFF
        shuf = randomAddr(shuf);

        PBA = (shuf << SHAMT) + (LBA & ~MASK);

        return PBA;
}

uint32_t randomAddr(uint32_t max_num) {
        int result = 0, min_num = 0, low_num = 0, hi_num = 0;
```

```
    if (min_num < max_num)
    {
        low_num = min_num;
        hi_num = max_num + 1; // include max_num in output
    } else {
        low_num = max_num + 1; // include max_num in output
        hi_num = min_num;
    }

    result = (rand() % (hi_num - low_num)) + low_num;
    return result;
}


void ptoa(uint8_t *addr, uint32_t PBA) {
         addr[0] = (uint8_t)(PBA & 0x000000FF);
         addr[1] = (uint8_t)(PBA & 0x00001F00)>>8;
         addr[2] = (uint8_t)(PBA & 0x001FE000)>>13;
         addr[3] = (uint8_t)(PBA & 0x1FE00000)>>21;
         addr[4] = (uint8_t)(PBA & 0x60000000)>>29;
}

void Collect_Garbage(void){
         uint8_t addr[5], status = 0;
         uint32_t PBA;
         while(top != NULL) {
                  PBA = pop();
                  ptoa(&(addr[0]),PBA);
                  do {
                           status = Block_Erase(&(addr[0]));
                  } while((status&0x21) != 0x21);
         }
}

void push(uint32_t item) {
    struct Node *temp;
    temp = (struct Node *)malloc(sizeof(struct Node));
    temp->val = item;
    if(top == NULL)
        temp->next = NULL;
    else
        temp->next = top;
    top = temp;
}

uint32_t pop() {
    if(top == NULL)
        return -1;
    else {
        uint32_t retval;
        struct Node *temp = top;
        retval = temp->val;
        top = temp->next;
        free(temp);
        return retval;
    }
}
```

```
//-----------------------------------------------------------------------------
// F34x_MSD_FTL.h
//-----------------------------------------------------------------------------
#include <stdio.h>

#ifndef __FLASH_BASIC_H__
#define __FLASH_BASIC_H__

#include <C8051F340.h>
#include <intrins.h>
#include <stdio.h>
#include <stdint.h>

sbit WP =         P3^6;    // Active low
sbit WE =         P3^5;    // Active low
sbit ALE =        P3^4;    // Active high
sbit CLE =        P3^3;    // Active high
sbit CE =         P3^2;    // Active low
sbit RE =         P3^1;    // Active low
sbit RYBY =       P3^7;

//sbit DQ =       P2;
```

```
#define FLASH_CMD(value) {\
        P3 &= ~(0x14);\
        CE = 0;\
        CLE = 1;\
        FLASH_DQ_LATCH_IN(value);\
        CLE = 0;\
}\

#define FLASH_ADDR_INPUT(value,max) {\
        int i;\
        CE=0;\
        CLE=0; ALE=1;\
        for(i=0;i<max;i++) {\
                FLASH_DQ_LATCH_IN(value[i]);\
        }\
        ALE = 0;\
}

// Read from flash
#define FLASH_DQ_LATCH_OUT(value) {\
        CE = 0;\
        P2MDOUT |= 0x00;\
        P2 |= 0xFF;\
        RE = 0;\
        value = P2;\
        RE = 1;\
}

// Write to flash
#define FLASH_DQ_LATCH_IN(value) {\
        CE = 0;\
        P2MDOUT &= 0xFF;\
        WE = 0;\
        P2 = value;\
        WE = 1;\
}\

#define FLASH_WAIT_BUSY {\
        while(!RYBY);\
}\

#define FLASH_STATUS_READ(value) {\
        FLASH_CMD(0x70);\
        FLASH_DQ_LATCH_OUT(value);\
}\

#define FLASH_IDLE {\
        CE = 1;\
}\

void Init_Flash(void);
void Page_Read(uint8_t * addr, uint8_t * buf);
uint8_t Page_Program(uint8_t * addr, uint8_t * buf);
uint8_t Block_Erase(uint8_t * addr);
void Page_Copy(uint8_t old_addr, uint8_t new_addr);
void Reset(void);
void ID_Read(uint8_t * str);
void Power_On(void);

#ifndef MACRO_FLASH_VERSIONS

#endif
#endif
```